ON LOCATING THE SIMPLE CYCLES IN A DIGRAPH

John Mackay Cochrane

United States
Naval Postgraduate School

THESIS

ON LOCATING THE SIMPLE CYCLES IN A DIGRAPH

by

John Mackay Cochrane

June 1970

On Locating the Simple Cycles in a Digraph

by

John Mackay Cochrane
Lieutenant (junior grade), United States Navy
B.S., United States Naval Academy, 1969

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June 1970

ABSTRACT

An algorithm is stated for finding the simple cycles in a digraph which is believed to be superior to previous algorithms. The algorithm is stated in a way which lends itself to use on a digital computer. Suitable modifications are presented which allow the algorithm to be applied to coalesced graphs. Finally, the algorithm is compared to a previously used technique, and is shown to require fewer operations.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

The author is grateful for the inspiration and guidance provided by Associate Professor Uno R. Kodres.

I. INTRODUCTION

One of the important problems in the theory of graphs, especially in its applications to physical systems, is that of locating the simple cycles in a directed graph. The author conducted an extensive literature search (all American Mathematical Society Reviews, I.E.E.E. Circuit Theory Reviews, and American Computing Machinery Reviews), and was unable to find any really efficient methods for solving the problem. One of the more efficient methods was the algorithm of Danielson [3], which uses matrix techniques and is designed for application to undirected graphs, although it lends itself to directed graphs if certain modifications are made. In comparison with the algorithm which we shall develope, it will be seen that it is itself rather inefficient.

We will first give a number of fundamental concepts, with examples, in order to familiarize the reader with the basics involved, and in order to make the paper as nearly self contained as possible. In this regard, we have adapted considerable material from Busacker and Saaty [2].

After a consideration of the basic theory, we develope an algorithm for locating the simple cycles in a directed graph. In doing so, we state some specific definitions which may not be found in the literature, but which are fundamental to our algorithm. After presenting the algorithm, we give an example to show how it works, and then we prove that the algorithm does indeed do what it is supposed to do.

Following the statement of the algorithm, we give modifications which allow the algorithm to be applied to coalesced graphs (graphs in which a subgraph is represented by a single vertex. We then give an example, which is taken from circuit theory, and show how the algorithm is applied.

II. FUNDAMENTAL CONCEPTS

Let S be a set. Suppose s is an element of S, and that t is an element of S. By the symbol (s&t) we mean the unordered pair s and t (unordered in the sense that (s&t) and (t&s) denote the same element). We shall use the symbol (S&S) to denote the unordered product of S with itself. Now, let V be any set which is non-empty, and let E be a set disjoint from V. Define a mapping g from E into (V&V). An undirected graph is the triple (V,E,g). We denote such a graph by

$$G = G(V,E,g),$$

or, simply

$$G = G(V,E)$$

if the mapping is understood. We call the elements of V vertices, the elements of E edges, and g is called the incidence mapping for G.

A directed graph is the triple

$$G = G(V,E,g'),$$

where V and E are defined as above and where g' maps E into (VxV), the unusual Cartesian product of V with itself.

We use the terminology of Harary [4] and call a directed graph a digraph. As in the undirected case, we call the elements of V vertices and the elements of E edges. We shall denote both elements of V and elements of E by subscripted lower case letters. If g' is one-to-one, we shall sometimes denote an element e of E by the pair (v_1,v_2) of vertices for which $g'(e) = (v_1,v_2)$. Note that to each digraph there corresponds an undirected graph formed from the digraph by removing the direction from all edges.

A graph

$$H = H(V_H,E_H,g_H)$$

is said to be a subgraph of a graph $G = G(V,E,g)$ if and only if the following conditions hold:

$$\text{(i)}\ V_H \subseteq V,$$

$$\text{(ii)}\ E_H \subseteq E,$$

$$\text{(iii)}\ g_H = g\Big|_H ,$$

where $g\Big|_H$ denotes the restriction of g to the subset E_H of E.

If e is an edge in a graph $G = G(V,E,g)$, and if

$$g(e) = (v\&w)$$

(or (v,w) or (w,v) in the case of a digraph), then e is said to be incident to v and w. If the graph is directed, then an edge e such that

$$g(e) = (v,w)$$

is said to be incident into w and incident out of v. Alternately, v is referred to as the initial vertex of e and w is referred to as the final vertex of e. A vertex v in a digraph whose incident edges are incident out of v is called a source; a vertex w whose incident edges are incident into w is called a sink.

Geometrically, we represent an undirected graph by associating each vertex with a geometrical point and each edge with a curve segment. A digraph is represented in the same way, except that the curve segment is directed according to the order of the vertices corresponding to the edge. Figure 1(a) shows a geometrical representation of an undirected graph, while Figure 1(b) shows a digraph.

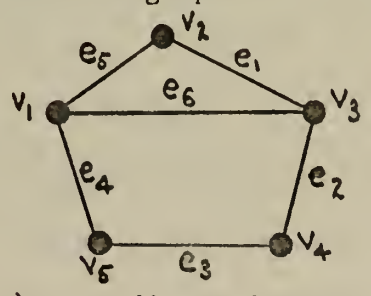
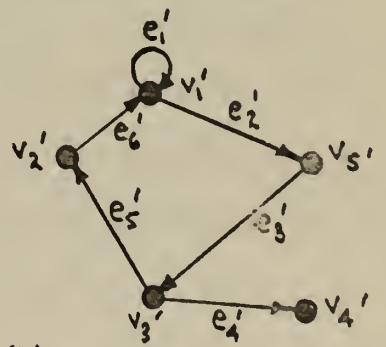


Figure 1(a) an undirected graph



(b) a digraph

For the graph in Figure 1(a),

$$V = \{v_1, v_2, v_3, v_4, v_5\},$$

$$E = \{e_1, e_2, e_3, e_4, e_5, e_6\},$$

$$g: e_1 - (v_2 \& v_3),\ e_2 - (v_3 \& v_4),\ e_3 - (v_4 \& v_5),\ e_4 - (v_1 \& v_5),$$

$$e_5 - (v_1 \& v_2),\ e_6 - (v_1 \& v_3).$$

For the graph in 1(b),

$$V' = \{v_1', v_2', v_3', v_4', v_5'\},$$

$$E' = \{e_1', e_2', e_3', e_4', e_5', e_6'\},$$

$$g': e_1' - (v_1', v_1'),\ e_2' - (v_1', v_5'),\ e_3' - (v_5', v_3'),\ e_4' - (v_3', v_4')$$

$$e_5' - (v_3', v_2'),\ e_6' - (v_2', v_1').$$

In the sequel, we shall consider only digraphs.

In a digraph, two edges are said to be consecutive provided the initial vertex of one is the final vertex of the other. Thus, two edges e_1 and e_2 such that

$$g(e_1) = (a,b) \text{ and } g(e_2) = (b,c)$$

are consecutive edges. A sequence of distinct consecutive edges of the form

$$P = (e_1, e_2, e_3, \ldots, e_{n-1})$$

such that

$$g(e_1) = (v_1, v_2),\ g(e_2) = (v_2, v_3),\ g(e_3) = (v_3, v_4), \ldots, g(e_{n-2}) = (v_{n-2}, v_{n-1})\ \ g(e_{n-1}) = (v_{n-1}, v_n)$$

is called a path. Note that no edges are repeated in a path, however, vertices may be repeated. A subsequence P' of consecutive edges is called a subpath of the path P. If P' starts with the same edge e_1 as P, we say P' is an initial subpath of P. We shall say a vertex v_i is attainable from a vertex v_o provided there is a path from v_o to v_i.

Alternately, we say that v_i is a descendant of v_o and that v_o is a predecessor of v_i.

We define a function d on the set $\mathcal{P}$ of all paths in a graph G into the set of natural numbers (including zero) by

$$d(P) = n, \quad P \text{ in } \mathcal{P},$$

where n is the number of edges in the path P. d(P) is referred to as the length of the path P. Note that an edge is a path of length one.

Let G = G(V,E,g) be a digraph. Define a multi-valued function f on V into V according to the following rule:

$$f(v) = \{v' : v' \text{ in } V \text{ and there exists } e \text{ in } E \text{ such that } g(e) = (v,v')\},$$

for all v in V. We call f the descendent function of the vertex v. We define powers of f as follows:

$$f^j(v) = \{v' : v' \text{ in } V,\ v' \text{ in } D_{vj}\}, \quad v \text{ in } V,$$

where D_{vj} is the set of all descendants of v by a path of length j. For example, in Figure 1(b), we have

$$f(v_1') = \{v_1', v_5'\}, \quad f^4(v_1') = \{v_1', v_2', v_4'\}.$$

A special type of path in a digraph is one which starts at a vertex v_o and ends at a vertex v_n, where $v_o = v_n$. Such a path is called a cycle. If no vertex (except v_o) is used more than once in a cycle, then the cycle is said to be simple. A simple cycle of length one is called a loop. A graph which contains no cycles is referred to as acyclic.

Suppose G = G(V,E,g) is an acyclic graph in which all vertices are attainable from a single vertex v_o in V. Further, suppose no vertex has more than one edge incident into it. Then G is called a rooted ditree. v_o is called the root of G. Figure 2 below shows a rooted ditree

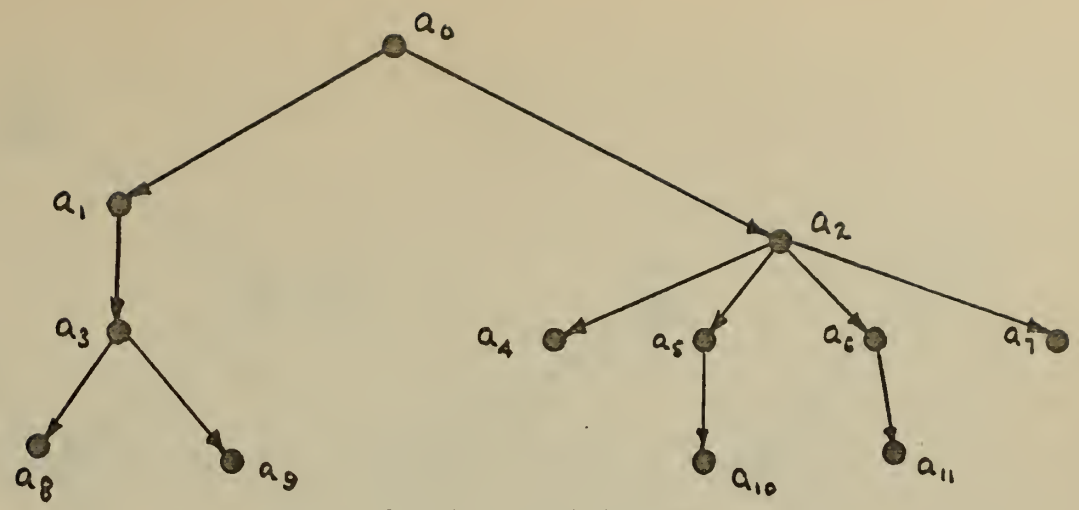


Figure 2. A rooted ditree.

Note that a rooted ditree must have only one root. A vertex in a rooted ditree is said to be terminal if it has no descendants. Thus, in the graph of Figure 2, a_0 is a root and $a_8, a_9, a_4, a_{10}, a_{11}, a_7$ are terminal vertices. A branch in a rooted ditree is any path whose initial vertex is the root and whose final vertex is terminal. Hence, the sequence of edges

$$B = ((a_0,a_2),(a_2,a_5)(a_5,a_{10}))$$

(note that since there are no multiple edges between any pair of vertices, there is no confusion if we represent an edge by its associated pair of vertices) is a branch of the ditree in Figure 2.

A graph $G = G(V,E,g)$ whose vertex set V can be partitioned into two non-null disjoint sets V_1 and V_2 such that E consists only of edges e_i such that

$$g(e_i) = (v_i,v_j) \text{ or } (v_j,v_i),\ v_i \text{ in } V_1,\ v_j \text{ in } V_2,$$

is called a bipatite directed graph or bi-digraph. The graph in Figure 3 is a bi-digraph.

Note that

$$V = \{a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9\}$$
$$= \{a_1,a_4,a_6,a_9\} \cup \{a_2,a_3,a_5,a_7,a_8\} = V_1 \cup V_2.$$

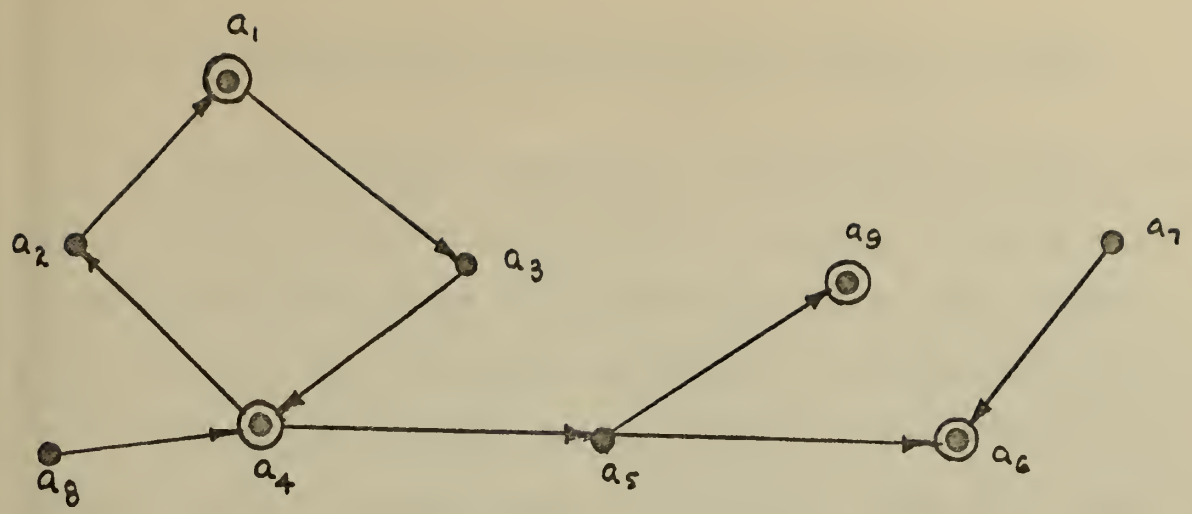


Figure 3. A bi-digraph

We have the following result concerning bi-digraphs.

THEOREM I: If G is a bi-digraph, then all simple cycles in G are of even length.

Proof: Suppose C is a simple cycle in G of the form

$$C = (e_1, e_2, \ldots, e_{n-1}, e_n),$$

where $d(C)$ is odd. If the length of C is odd, then the initial vertex of C must be different from the final vertex by the definition of bi-partite graph. But this is impossible since the initial and final vertex of a cycle must coincide. Therefore, $d(C)$ is even.

III. AN ALGORITHM FOR FINDING THE SIMPLE CYCLES IN A DIGRAPH

An important problem, especially in the application of graph techniques, is that of locating simple cycles in a digraph. In dealing with large graphs (large in the sense that V and E contain many elements), it is useful to have an efficient algorithm which conserves memory space of a computer. The algorithm we shall develope in this chapter is designed to meet these requirements. In the final chapter of the paper, we will show how our algorithm compares with an algorithm given by Denielson [3].

Before starting, it will be necessary to introduce some new terminology. In the sequel, we shall denote the vertices of all graphs under consideration by integer-indexed lower case letters such that no two vertices have the same index. If j is an integer such that v_j occurs in a graph under consideration, we will say v_j is the vertex corresponding to the indexing integer j.

DEFINITION 1.

Let G = G(V,E,g) be a digraph. Let v_i be an element of V. Then, the indexing integer j corresponding to v_j is said to be an adjacent element of v_i provided v_j belongs to $f(v_i)$, where f is the descendent function of v_i.

For example, in the graph in Figure 3 above, the adjacent elements of v_5 are the integers 6 and 9.

DEFINITION 2.

If G = G(V,E,g) is a digraph, and if v_i is a vertex in V, then the adjacency set of v_i is the set of all adjacent elements of v_i (e.g., the adjacency set of v_i is the set of index-integers corresponding to the elements of $f(v_i)$).

For example, in the graph in Figure 3, the adjacency set of a_5 is the set $\{6,9\}$. If no vertices are adjacent to a vertex (i.e., if a vertex is a sink) we denote its adjacency set by $\emptyset$ (the null set).

In the discussion which follows, we assume that the graph under consideration, G = G(V,E,g) is a digraph on n vertices which are represented by the lower case letter v indexed 1 through n. We further assume that in order to communicate to the computer the structure of G, we have a list which associates each vertex in V with its adjacency set.

DEFINITION 3.

An attainability chain of a vertex v_i in a digraph G is a directed graph C_i, which corresponds to a simple path $P(v_i,v_j)$ in G. The source vertex of C_i is i (corresponding to v_i), and the sink vertex is either a sink vertex in G, or a copy k' of the first repeating vertex v_k in the path.

For example, consider the graph in Figure 3. The vertex a_1 has the following attainability chains:

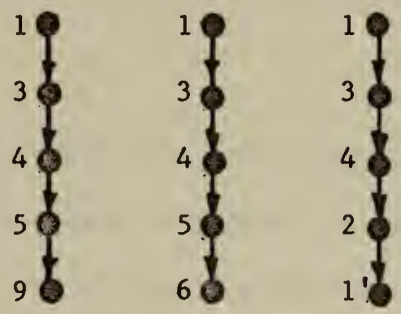


Figure 4. Attainability chains for the vertex a_1 of the graph in Figure 3. Note that if a simple cycle occurs, as in the cycle composed of the vertices a_1, a_3, a_4, a_2, a_1, then the attainability chain of any of these vertices terminates as soon as the first repetition of a vertex occurs. Also, note that each attainability chain is a rooted ditree with one branch.

DEFINITION 4.

The attainability tree for a vertex v_i is a rooted ditree with a vertex corresponding to i in the root position and with branches consisting of all the attainability chains of v_i. We will super-impose the initial subpaths of any attainability chains with identical initial subpaths.

The graph in Figure 5 shows the attainability tree for the vertex a_1 of Figure 3.

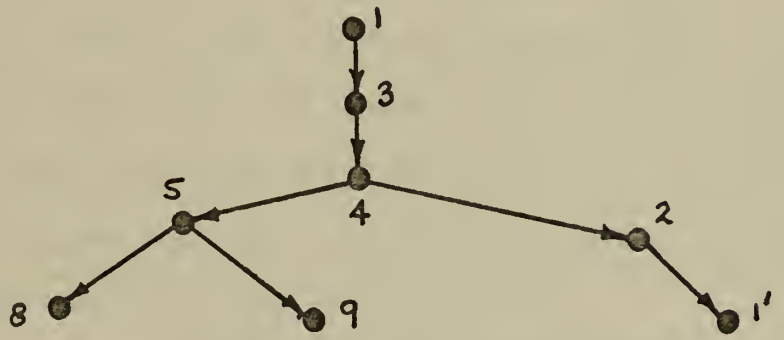


Figure 5. An Attainability Tree

We now give the formal statement of the algorithm, keeping in mind that the graph under consideration is a digraph G = G(V,E,g) on n vertices, and that we have a list of vertices and their corresponding attainability sets.

STEP 1. Inspect the list and eliminate all vertices whose index integers do not appear as adjacent elements. Obviously, these vertices could not belong to simple cycles because they have no edges incident into them (i.e., they are sources). Also, inspect the list and eliminate as roots all vertices whose adjacency set is Ø. Again, it is clear that such vertices cannot belong to any simple cycles because there are no edges incident out of them (i.e., they are sinks). We have thus eliminated all sources and

sinks as possible roots. We may now repeat the procedure on the remaining vertices until no more eliminations are possible.

STEP 2. Pick any vertex v_i from the list which has not been eliminated and construct its attainability tree, consulting the origional list for incidence relations. Recall that we superimpose identical initial subpaths.

STEP 3. If an element is repeated along any attainability chain, record that element together with the elements between the two occurrence (in the order they appear). The vertices corresponding to these elements will compose a simple cycle in G. Note that repeated vertices must occur along the same branch of the attainability tree for a cycle to be located.

STEP 4. Eliminate as possible roots all elements which appeared in the attainability tree of v_i, for if these vertices belonged to simple cycles, this would have been discovered in the attainability tree of v_i.

STEP 5. If all vertices have been eliminated as possible roots, the algorithm terminates. If some vertices remain, choose any such vertex v_k not eliminated as a possible root, construct its attainability tree and return to step 3.

We now ask the following question: Does the algorithm locate all the simple cycles in a digraph (and only the simple cycles)? We answer this question with the following theorem:

THEOREM II. The algorithm outlined in Steps 1 through 5 above locates all of the simple cycles of a digraph, and it locates only simple cycles.

Proof: First, note that no cycles will be located which are not simple, since a branch is terminated at the first repetition of a vertex.

Now, suppose there is a simple cycle C in the digraph G which was not located by the algorithm. Then C can have no edges in common with any attainability tree considered in the sequence, for if it did, then by the fact that a branch is terminated only if a source, a sink, or a repetition occurs, C would have been located. Conversely, only those edges of G which involve sources or sinks in G are left out of the set of attainability trees. Thus, we have established that if C is not located by the algorithm, then C does not belong to G, which is a contradiction. Therefore, C must have been located by the algorithm.

It should be noted at this point that some cycles may be located more than once. However, this will occur only in special cases.

As an example of the application of this algorithm, consider the digraph shown in Figure 6 below.

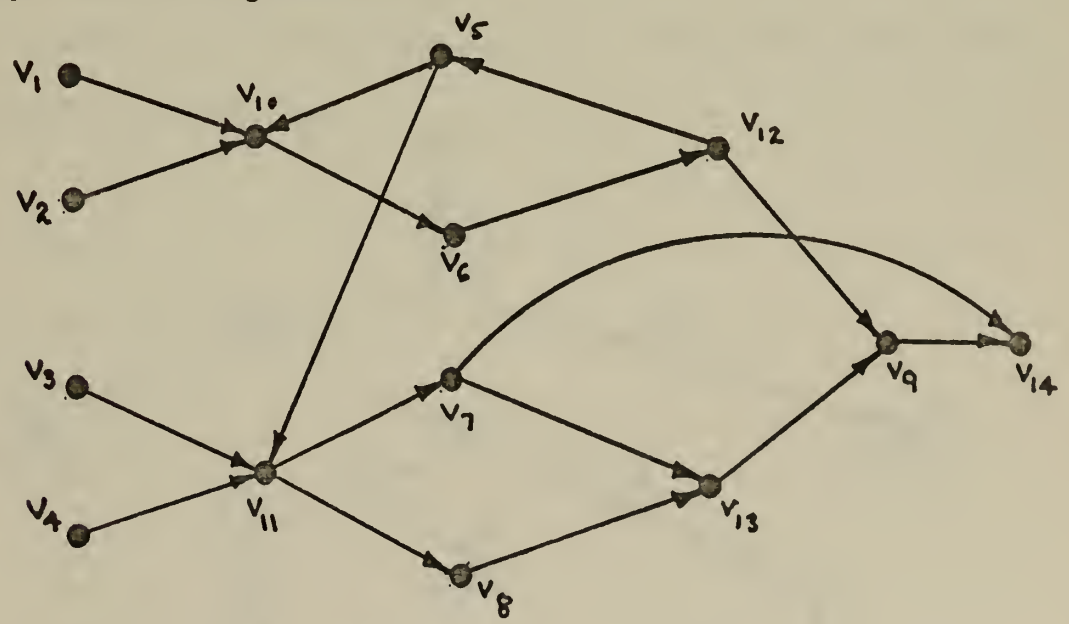


Figure 6. A Digraph

We have the following list of vertices with their adjacency sets:

v_1: {10}
v_2: {10}
v_3: {11}
v_4: {11}
v_5: {10, 11}
v_6: {12}
v_7: {13, 14}
v_8: {13}
v_9: {14}
v_{10}: {6}
v_{11}: {7, 8}
v_{12}: {5, 9}
v_{13}: {9}
v_{14}: ∅

Applying Step 1, we eliminate v_{14} as a possible root, since its adjacency set is ∅. We also eliminate v_1, v_2, v_3, and v_4, since 1, 2, 3, and 4 do not occur in any of the adjacency sets. Let us now construct the attainability tree of v_{10}.

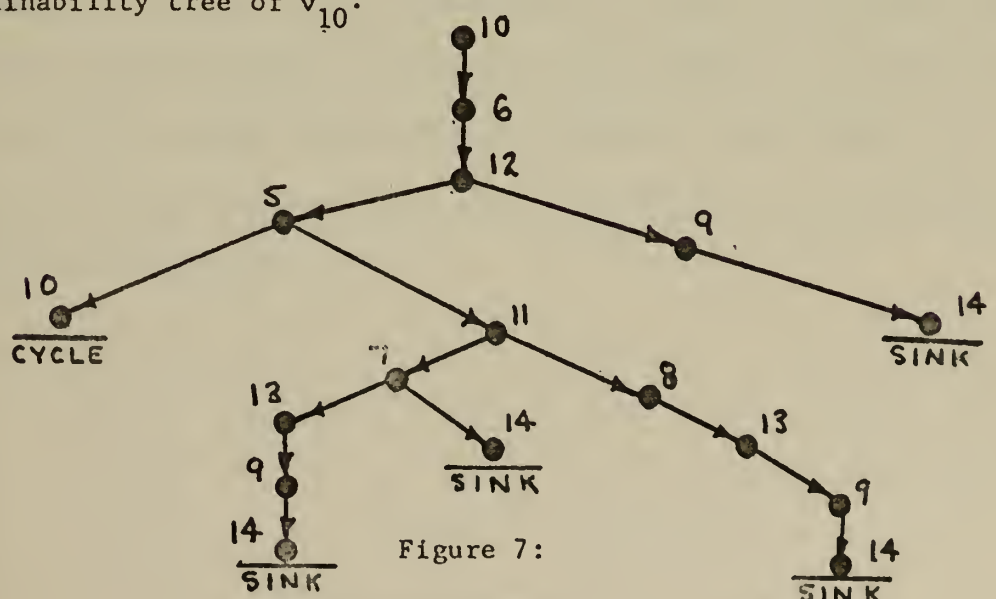


Figure 7:
Attainability tree for v_{10} of Figure 6

We have located one simple cycle, namely

$$C = \left((v_{10},v_6),(v_6,v_{12}),\ (v_{12},v_5),(v_5,v_{10})\right) .$$

Note that 5, 6, 7, 8, 9, 10, 11, 12, 13 all appeared in the attainability tree of v_{10}, and that 1, 2, 3, 4, and 14 have already been eliminated as roots. Hence, by Step 4, we need construct no more attainability trees, and we can conclude that C is the only simple cycle in the graph. An inspection of Figure 6 also leads to this conclusion.

Note that if the graph under consideration were a bipartite digraph (or bi-digraph), we would have had two lists instead of one. In this case, we would add the following step the simplifications above:

> STEP 0: Eliminate as possible roots all vertices which occur in the list with the most vertices. This will be sufficient to locate all simple cycles by Theorem I which states that all cycles in a bi-digraph have even length, and hence must contain at least one vertex from each list. This simplification will be used in the next chapter.

We observe at this point, that though the primary purpose of our algorithm as stated is to locate simple cycles, it may also be used to locate any simple path in a digraph. For example, if we wish to know if there is a simple path between a vertex v_0 and a vertex v_i in a digraph, we need only construct the attainability tree of v_0 and note whether it contains v_i.

IV. COALESCED GRAPHS

The purpose of the foregoing algorithm is to make possible the location of all simple cycles in a digraph with a large number of vertices. In practice, in the analysis of very large graphs, such as the graph of the circuitry of a digital computer, a procedure is used which partitions the graph into subgraphs which are, in turn, partitioned into smaller subgraphs, until manageable subgraphs are obtained. Once these subgraphs have been analyzed for cyclic structure, they are generally coalesced into a single vertex, and the resulting graph is then analyzed for its cyclic structure. In this chapter, we will show how the technique of Chapter II may be applied to such "coalesced graphs."

One of the principal problems in locating the simple cycles in a coalesced graph is that a vertex which is a coalesced subgraph may be attainable from a vertex v, but a vertex v* which is attainable from the coalesced subgraph may <u>not</u> be attainable from v in the original graph. For example, suppose G is the graph shown in Figure 7.

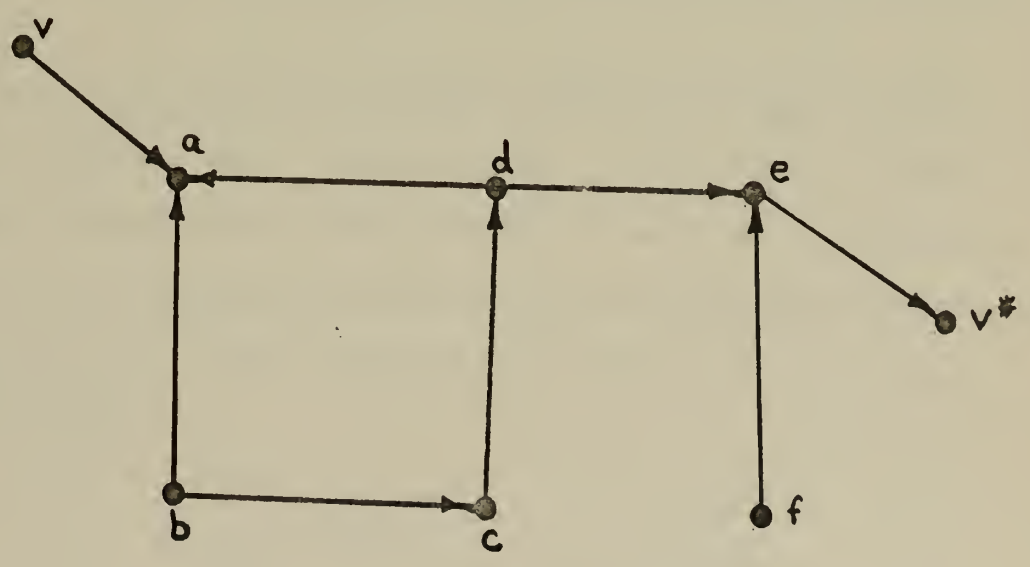


Figure 7. A Digraph

Suppose that, by some procedure, we determine that the subgraph of G composed of the vertices a, b, c, d, e, and f, together with the edges connecting them, can be coalesced into a single vertex v_c. Then, the graph shown in Figure 8 is the coalesced version of the graph in Figure 7.

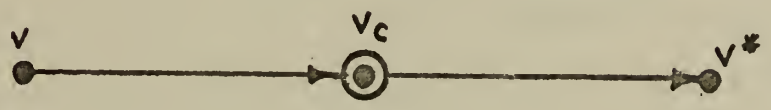


Figure 8. The Coalesced Version of the Graph in Figure 7

Note that this graph is bipartite, since one node actually represents a subgraph, while the others represent single vertices in the original graph. Also, note that in the graph in Figure 8, v* is a descendant of v but, in the original graph (Figure 7), it is not. Thus, we see that our algorithm cannot be applied to coalesced graphs without some modification.

Let $G_o = G_o(V_o,E_o,g_o)$ be a digraph without coalesced vertices. Let S be a subgraph of G_o. Form the graph $G_c = G_c(V_c,E_c,g_c)$ in which S has been coalesced into a single vertex.s. Let f_o be the descendent function on the vertices of G_o, and let f_c be the descendent function on the vertices of G_c. Define subsets I(S) and J(S) on the vertex set V_c of G_c as follows:

$$I(S) = \{v_i \text{ in } V_c\text{: } s \text{ is in } f_c(v_i)\}$$

$$J(S) = \{v_j \text{ in } V_c\text{: } v_j \text{ is in } f_c(s)\}$$

Now, define a mapping p on I(S) x J(S) into 0,1 such that

$$p(v_i v_j) = \begin{cases} 0 \text{ if } v_j \text{ is not in } f_o^k(v_i) \text{ for all } k, \\ 1 \text{ if there exists } k \text{ such that } v_j \text{ is in } f_o^k(v_i). \end{cases}$$

Compute $p(v_i,v_j)$ for all pairs (v_i,v_j) with v_i in I(S) and v_j in J(S) as follows: Let s_i be in S such that s_i is also in $f_o(v_i)$. Let s_j be such that v_j is in $f_o(s_j)$. Note that s_i and s_j must exist since v_i is in I(S) and v_j is in J(S). Let T_{s_i} be the attainability tree of s_i, and suppose V_{s_i} is the vertex set of T_{s_i}. Then, if s_j is in V_{s_i},

$$p(v_i,v_j) = 1,$$

otherwise,

$$p(v_i v_j) = 0.$$

Note that T_{s_i} may not exist, in which case $p(v_i v_j)$ is obviously zero.

When $p(v_i,v_j)$ has been computed for all ordered pairs (v_i,v_j) with v_i in I(S) and v_j in J(S), we apply the algorithm of Chapter II to the bi-digraph G_c. Suppose we do this and obtain simple cycles $C_1, \ldots, C_n$. Note that by Theorem I, the length of C_i (i = 1, ... , n) will be even. Hence, if m_i is the number of vertices in C_i, then $m_i/2$ is the number of coalesced subgraphs in C_i. Denote these coalesced subgraphs by $k_{i1}, k_{i2}, \ldots, k_{im_i/2}$.

Let

$$I(k_{ij}) = \left\{v_m \text{ in } V_c : k_{ij} \text{ is in } f_c(v_n)\right\}, \; i = 1, \ldots, n; \; j = 1, \ldots, m_i/2$$

$$J(k_{ij}) = \left\{v_l \text{ in } V_c : v_l \text{ is in } f_c(k_{ij})\right\}, \; i = 1, \ldots, n; \; j = 1, \ldots, m_i/2$$

For each k_{ij} (i = 1, ..., n; j = 1, ..., $m_i/2$) form the ordered pair (t_{j-1},t_{j+1}), where t_{j-1} is in $I(k_{ij})$ and t_{j+1} is in $J(k_{ij})$ and both t_{j-1} and t_{j+1} are in C_i. Now, the ordered pair (t_{j-1}, t_{j+1}) must be in the

domain of p, since (t_{j-1},t_{j+1}) is in $I(k_{ij}) \times J(k_{ij})$. Note the value of $p(t_{j-1},t_{j+1})$. If it is zero for any j (j = 1, ... , $m_i/2$), then C_i is not actually a cycle in G_o. If it is one for all j, then C_i is indeed a cycle in G_o. For example, suppose G_o is the graph in Figure 9 below.

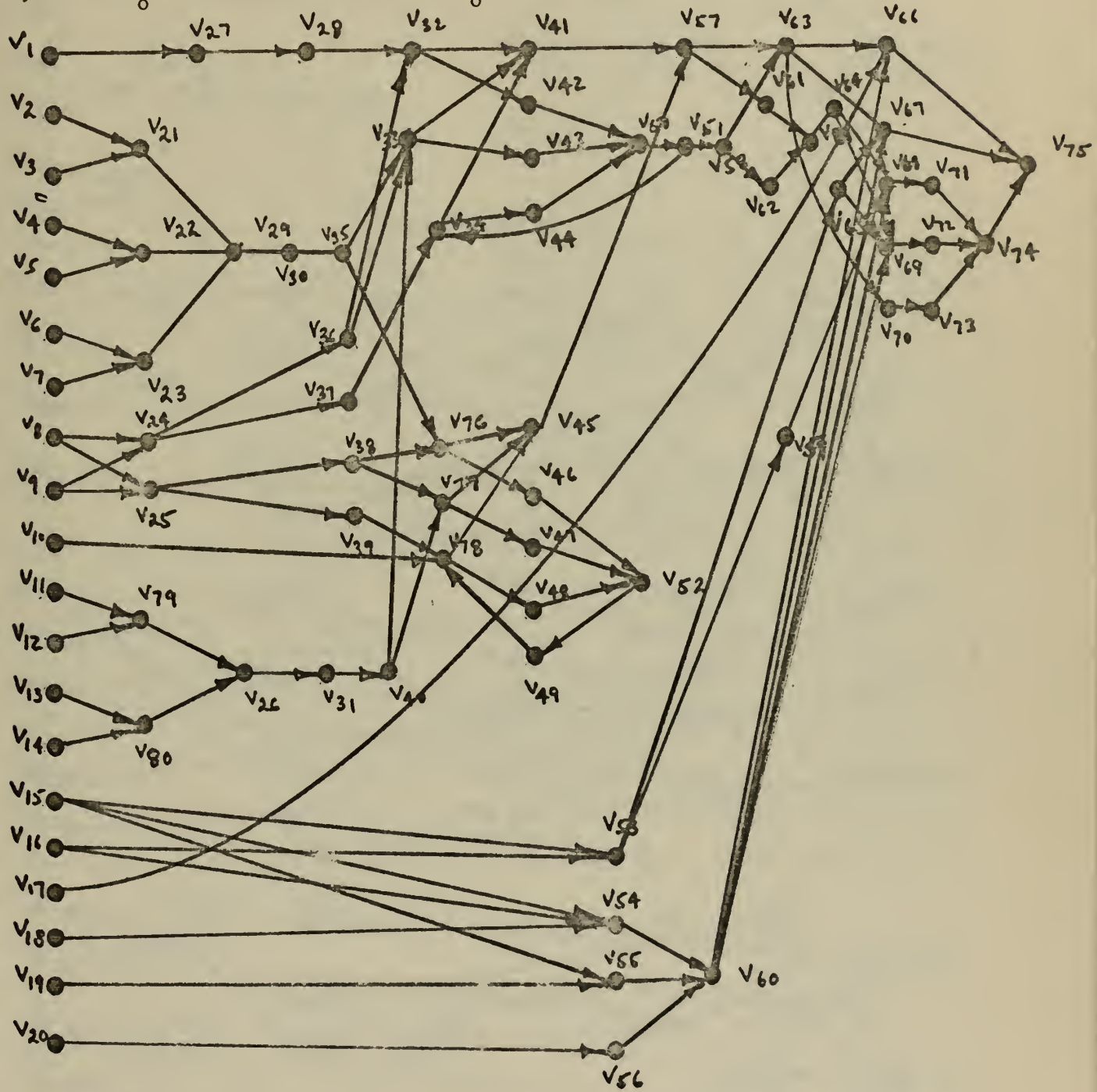


Figure 9. A Digraph

This is a digraph of an adder from a digital computer. Suppose we have a procedure which partitions G_o into the four subgraphs shown in Figure 10.

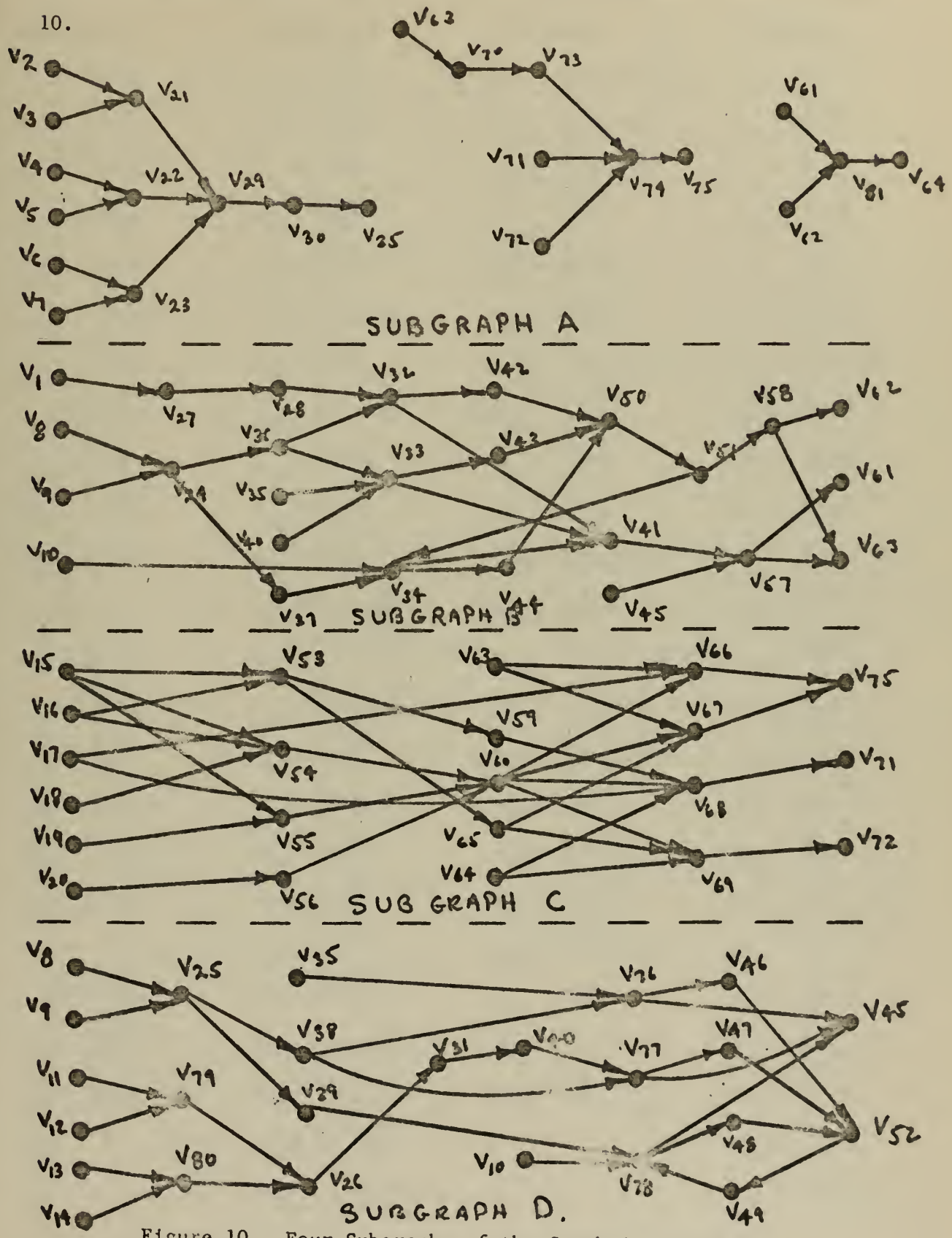


Figure 10. Four Subgraphs of the Graph in Figure 9.

We now apply our algorithm to the four subgraphs. We have the following lists of vertices together with their adjacency sets:

Subgraph A

v_2 : {21}
v_3 : {21}
v_4 : {22}
v_5 : {22}
v_6 : {23}
v_7 : {23}
v_{21}: {29}
v_{22}: {29}
v_{23}: {29}
v_{29}: {30}
v_{30}: {35}
v_{35}: ∅
v_{63}: {70}
v_{71}: {74}
v_{72}: {74}
v_{70}: {73}
v_{73}: {74}
v_{74}: {75}
v_{75}: ∅
v_{61}: {81}
v_{62}: {81}
v_{81}: {64}
v_{64}: ∅

Subgraph B

v_1 : {27}
v_8 : {24}
v_9 : {24}
v_{10}: {34}
v_{27}: {28}
v_{24}: {36.37}
v_{28}: {32}
v_{36}: {32,33}
v_{35}: {33}
v_{40}: {33}
v_{37}: {34}
v_{32}: {41,42}
v_{33}: {41,43}
v_{34}: {41,44}
v_{42}: {50}
v_{43}: {50}
v_{44}: {50}
v_{50}: {51}
v_{41}: {57}
v_{45}: {57}
v_{51}: {34,58}
v_{58}: {62}
v_{57}: {61,63}
v_{61}, v_{62}, v_{63}: ∅

Subgraph C

v_{15}: {53,54,55}
v_{16}: {53,54}
v_{17}: {66,68}
v_{18}: {54}
v_{19}: {55}
v_{20}: {56}
v_{53}: {59,65}
v_{54}: {60}
v_{55}: {60}
v_{56}: {60}
v_{63}: {66,67}
v_{59}: {68}
v_{60}: {66,67,68,69}
v_{65}: {67,69}
v_{64}: {68,69}
v_{66}: {75}
v_{67}: {75}
v_{68}: {71}
v_{69}: {72}
v_{75}: ∅
v_{71}: ∅
v_{72}: ∅

Subgraph D

v_8 : {25}
v_9 : {25}
v_{11}: {79}
v_{12}: {79}
v_{13}: {80}
v_{14}: {80}
v_{25}: {38,39}
v_{79}: {26}
v_{80}: {26}
v_{35}: {76}
v_{38}: {76,77}
v_{39}: {78}
v_{26}: {31}
v_{31}: {40}
v_{40}: {77}
v_{10}: {78}
v_{76}: {45,46}
v_{77}: {45,47}
v_{78}: {45,48}
v_{46}: {52}
v_{47}: {52}
v_{48}: {52}
v_{49}: {78}
v_{52}: {49}
v_{45}: ∅

We have the following attainability trees for the subgraphs A through D:

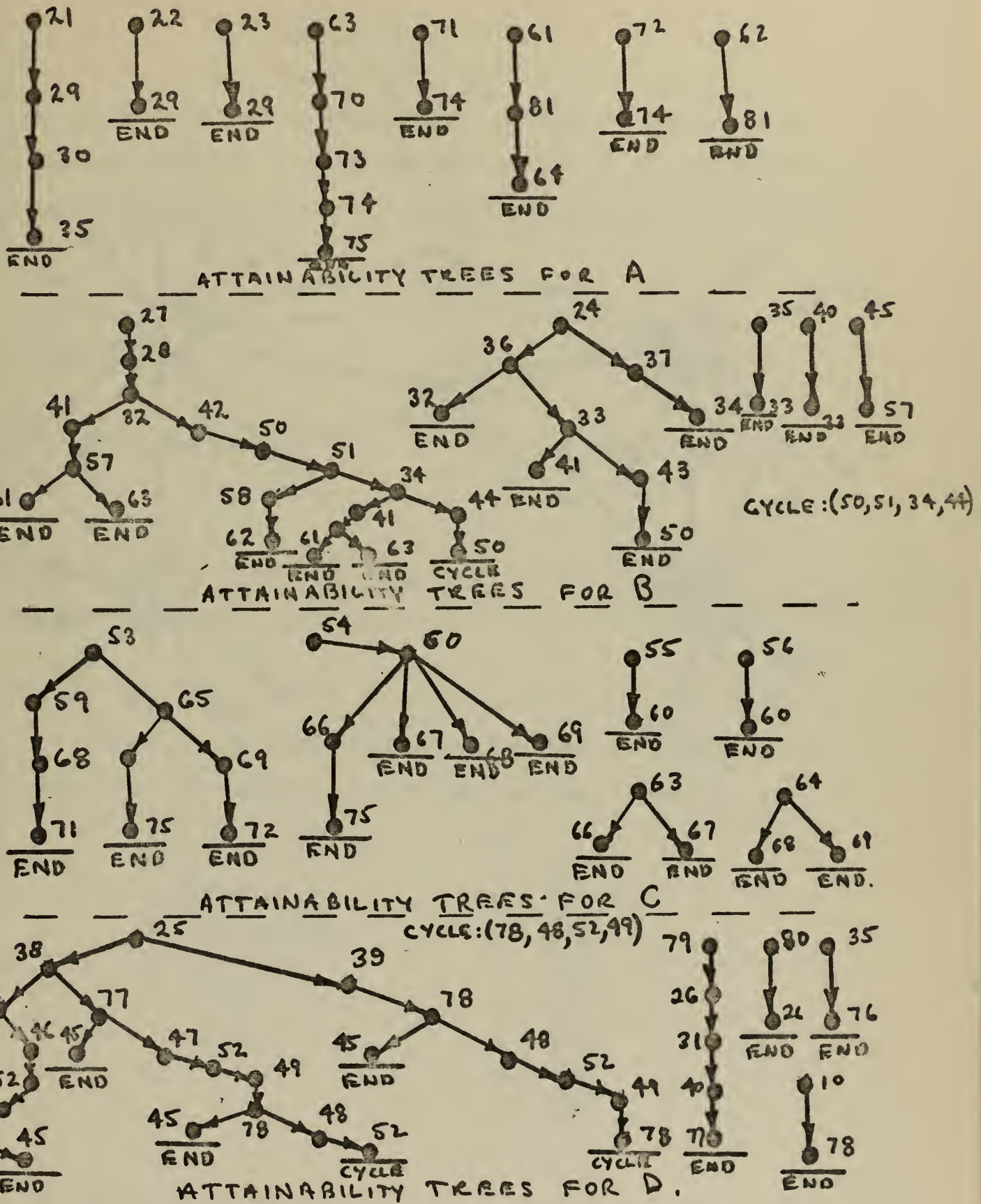


Figure 11. Attainability Trees for the Subgraphs of Figure 10.

We have located two simple cycles:

$$C_1 = (v_{50}, v_{51}, v_{34}, v_{44}, v_{50})$$

$$C_2 = (v_{78}, v_{48}, v_{52}, v_{49}, v_{78})$$

We now coalesce the four subgraphs into vertices and examine the resulting bi-digraph for cyclic structure in order to discover if there are any larger simple cycles in G_o. Figure 12 shows the coalesced graph G_c.

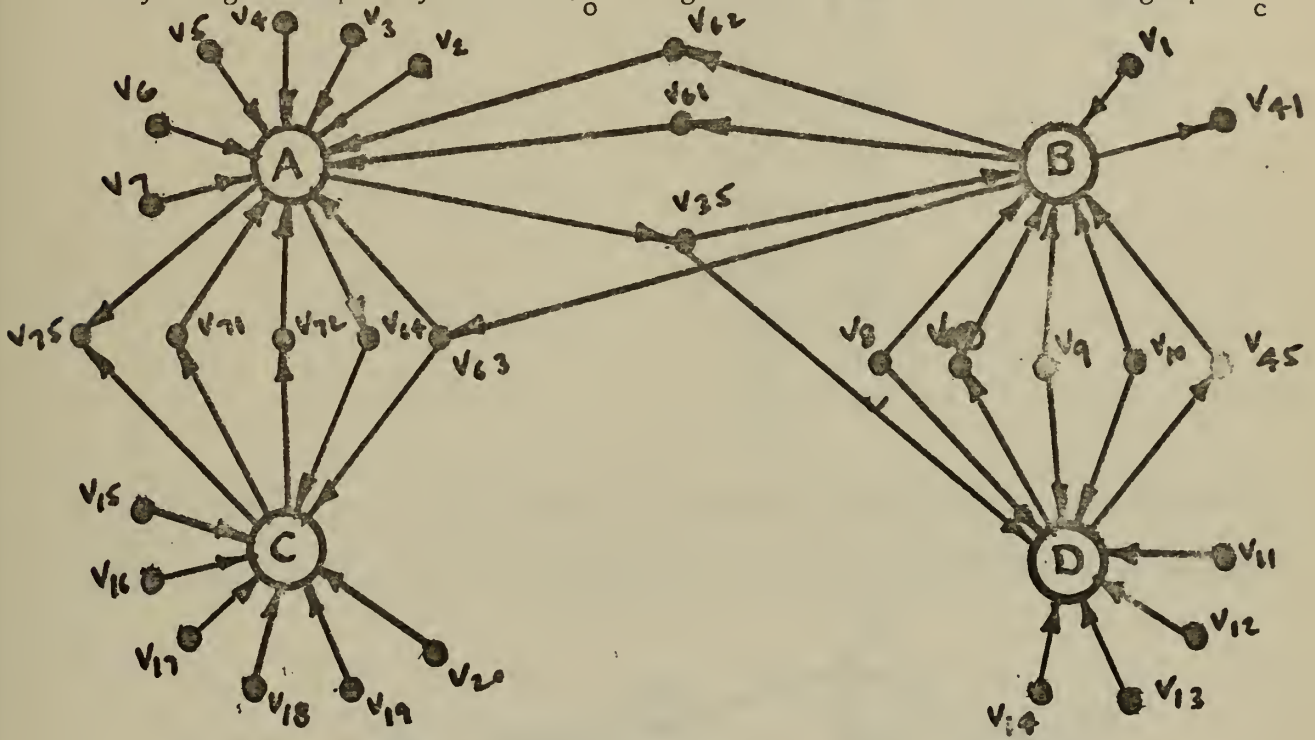


Figure 12. The Coalesced Graph of the Graph in Figure 9.

Since this is a bipartite graph, we have two lists of vertices:

V_1

A: {35,64,75}
B: {41,62,61,63}
C: {75,71,72}
D: {40,45}

V_2

v_2 : {A}
v_3 : {A}
v_4 : {A}
v_5 : {A}
v_6 : {A}
v_7 : {A}
v_{11}: {D}

V_2

v_{12}:	{D}	v_{71}:	{A}
v_{13}:	{D}	v_{72}:	{A}
v_{14}:	{D}	v_{64}:	{C}
v_{15}:	{C}	v_{63}:	{A,C}
v_{16}:	{C}	v_{8} :	{B,D}
v_{17}:	{C}	v_{40}:	{B}
v_{18}:	{C}	v_{9} :	{B,D}
v_{19}:	{C}	v_{10}:	{B,D}
v_{20}:	{C}	v_{45}:	{B}
v_{1} :	{B}		
v_{41}:	∅		
v_{62}:	{A}		
v_{61}:	{A}		
v_{35}:	{B,D}		
v_{75}:	∅		

We now invoke step 0 and consider as possible roots only those vertices which appear in list V_2. We construct the attainability tree for vertex A.

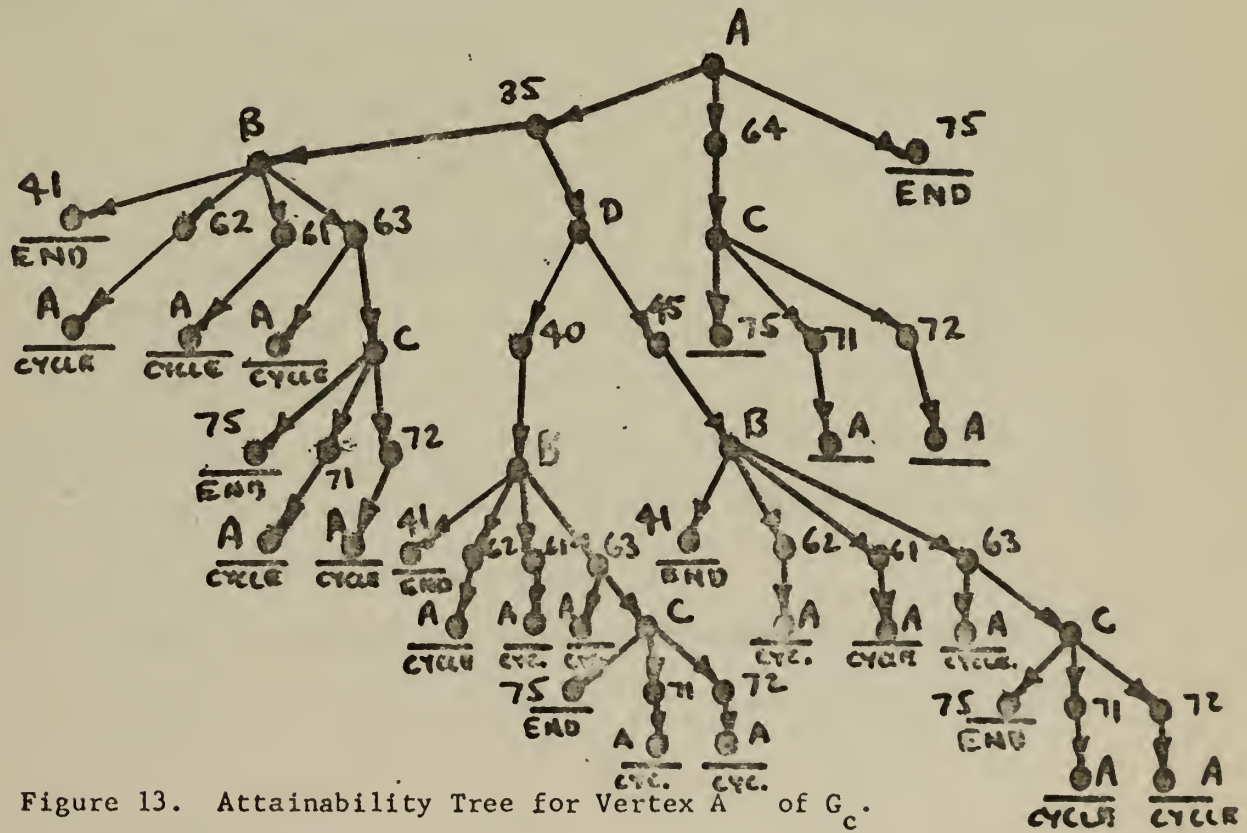


Figure 13. Attainability Tree for Vertex A of G_c.

We see that vertices B, C, and D, appear in the attainability tree of A, so it is unnecessary to construct any more attainability trees (by Step 4).

We have located the following simple cycles in G_c:

$C_1 = (A, v_{35}, B, v_{62}, A)$

$C_2 = (A, v_{35}, B, v_{61}, A)$

$C_3 = (A, v_{35}, B, v_{63}, A)$

$C_4 = (A, v_{35}, B, v_{63}, C, v_{71}, A)$

$C_5 = (A, v_{35}, B, v_{63}, C, v_{72}, A)$

$C_6 = (A, v_{35}, D, v_{40}, B, v_{62}, A)$

$C_7 = (A, v_{35}, D, v_{40}, B, v_{61}, A)$

$C_8 = (A, v_{35}, D, v_{40}, B, v_{63}, A)$

$C_9 = (A, v_{35}, D, v_{40}, B, v_{63}, C, v_{71}, A)$

$C_{10} = (A, v_{35}, D, v_{40}, B, v_{63}, C, v_{72}, A)$

$C_{11} = (A, v_{35}, D, v_{45}, B, v_{62}, A)$

$C_{12} = (A, v_{35}, D, v_{45}, B, v_{61}, A)$

$C_{13} = (A, v_{35}, D, v_{45}, B, v_{63}, A)$

$C_{14} = (A, v_{35}, D, v_{45}, B, v_{63}, C, v_{71}, A)$

$C_{15} = (A, v_{35}, D, v_{45}, B, v_{63}, C, v_{72}, A)$

$C_{16} = (A, v_{64}, C, v_{71}, A)$

$C_{17} = (A, v_{64}, C, v_{72}, A)$

We must now compute the values of p to see which of these simple cycles in G_c is also a simple cycle in G_o. We shall do this only for C_1. The others would be done exactly the same.

We first note that C_1 involves the coalesced vertices A and B, with A in $f_c(v_{62})$, v_{35} in $f_c(A)$, B in $f_c(v_{35})$, and v_{62} in $f_c(B)$. Hence,

we must compute $p(v_{62},v_{35})$ and$p(v_{35},v_{62})$. To compute $p(v_{62},v_{35})$, we examine subgraph A, and note that 62 does not appear in any of the attainability trees of the vertices of A. Hence, we must construct the attainability tree of v_{62} to see if v_{35} is a descendant of v_{62}:

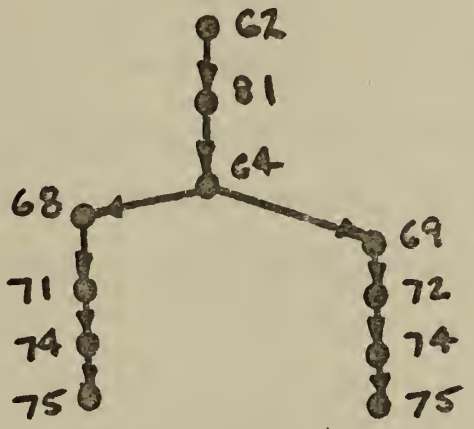


Figure 14. The Attainability Tree of v_{62}

We see that 35 does not appear in the attainability tree of $v_{62}$and thus, $p(v_{62},v_{35}) = 0$. Therefore, C_1 is not a cycle in G_o. We will not repeat the procedure for C_2 through C_{17}, however, if we did, we would find that none of these are cycles in G_o.

In actual applications of this technique, we would first determine the cyclic structure of each subgraph which is coalesced into a node. In addition to retaining the resulting information about cycles, we would also note $p(v_i,v_j)$ for each pair of nodes external to the coalesced subgraphs. We would then use this information in constructing the attainability trees for the coalesced graph. Had we done this above, the attainability tree in Figure 13 would have been less complicated. We did not follow this procedure above in order to better illustrate the use of the function p.

V. A COMPARISON WITH ANOTHER ALGOTHITHM

We shall now show how the algorithm developed in Chapter II for locating the simple cycles of a digraph is much superior, in terms of operations required, to another algorithm used for the same purpose.

Danielson [3] developed an algorithm for locating all paths (including cycles) in an undirected graph using matrix techniques. His method, while designed for undirected graphs, can easily be modified for application to digraphs. We now state the necessary definitions to explain his algorithm.

DEFINITION 5.

If $G = G(V,E,g)$ is an undirected graph on n vertices $(v_1, \ldots, v_n)$ and m edges $(e_1, \ldots, e_m)$, then the adjacency matrix A_o of G is given by $A_o = (a_{ij})$, where

$$a_{ij} = \begin{cases} 1 \text{ if an edge connects } v_i \text{ to } v_j, \\ 0 \text{ otherwise.} \end{cases}$$

DEFINITION 6.

If $G = G(V,E,g)$ is an undirected graph on n vertices $(v_1, \ldots, v_n)$ and m edges $(e_1, \ldots, e_m)$, then the variable adjacency matrix A of G is given by $A = (a_{ij})$, where

$$a_{ij} = \begin{cases} e_k \text{ if edge } e_k \text{ joins } v_i \text{ to } v_j, \\ 0 \quad \text{otherwise.} \end{cases}$$

DEFINITION 7.

If $G = G(V,E,g)$ is an undirected graph on n vertices $(v_1, \ldots, v_n)$, then the modified variable adjacency matrix B of G is given by $B = (b_{ij})$, where

$$b_{ij} = \begin{cases} v_j & \text{if an edge joins } v_i \text{ to } v_j, \\ 0 & \text{otherwise} \end{cases}$$

To illustrate these concepts, consider the graph of Figure 15.

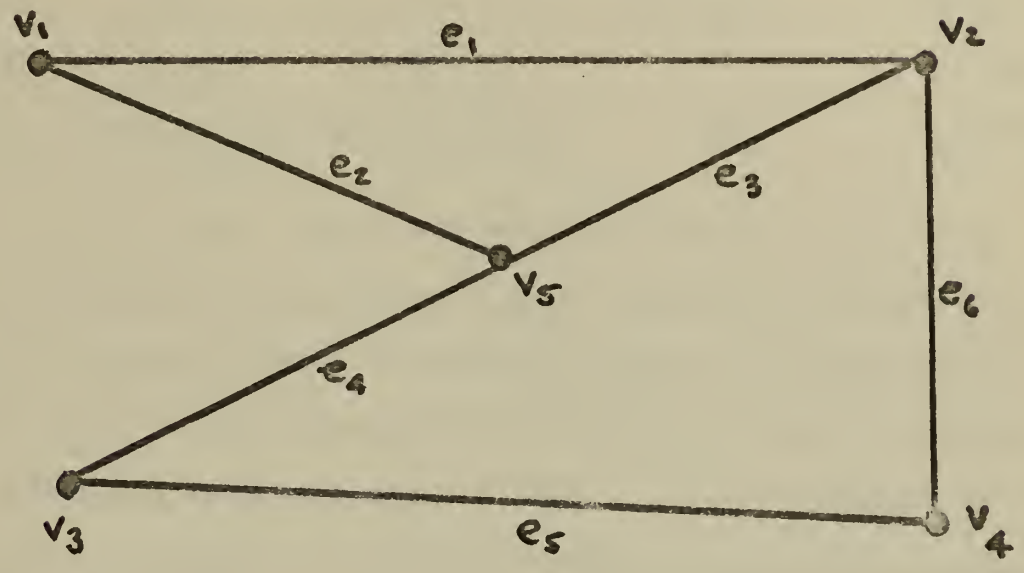


Figure 15. An Undirected Graph

We have

$$A_o = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & e_1 & 0 & 0 & e_2 \\ e_1 & 0 & 0 & e_6 & e_3 \\ 0 & 0 & 0 & e_5 & e_4 \\ 0 & e_6 & e_5 & 0 & 0 \\ e_2 & e_3 & e_4 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & v_2 & 0 & 0 & v_5 \\ v_1 & 0 & 0 & v_4 & v_5 \\ 0 & v_2 & v_3 & 0 & 0 \\ 0 & v_2 & v_3 & 0 & 0 \\ v_1 & v_2 & v_3 & 0 & 0 \end{bmatrix}.$$

The algorithm states that all edge progressions of length k between v_i and v_j will appear in the i-j position of P_k, where P_k is defined recursively as follows:

$$P_2 = BA_o, \quad P_k = BP_{k-1}.$$

Hence, for the graph in Figure 15,

$$P_3 = \begin{bmatrix} v_2v_5+v_5v_2 & v_5v_1 & v_2v_4+v_2v_5 & v_5v_2+v_5v_3 & v_2v_1 \\ v_4v_2+v_5v_2 & v_1v_5+v_5v_1 & v_1v_5 & v_1v_2+v_5v_2+v_5v_3 & v_1v_2+v_4v_2+v_4v_3 \\ v_4v_2+v_5v_2 & v_5v_1 & 0 & v_5v_2+v_5v_3 & v_4v_2+v_4v_3 \\ v_2v_5+v_3v_5 & v_3v_4+v_3v_5 & v_2v_4+v_2v_5 & 0 & v_2v_1 \\ v_2v_5+v_3v_5 & v_1v_5+v_3v_4+v_3v_5 & v_1v_5+v_2v_4+v_2v_5 & v_1v_2 & v_1v_2+v_2v_1 \end{bmatrix}.$$

We first note that, as stated before, the algorithm only locates the paths and cycles in an undirected graph. However, it is possible to modify the procedure slightly to obtain the cycles of a directed graph. How this is done is unimportant; what is important is that to locate all paths (including cycles) of lengths, k, k-1, ... , 2 in a digraph, we are required to multiply n x n matrices k times. In particular, if we were seeking all simple cycles in a complicated graph on n vertices, we would have no choice but to perform matrix multiplication n times. Now, in using this algorithm, we are not performing matrix multiplication in the usual manner, since the elements of the P_k's and of B are symbols (i.e., subscripted lower case letters) rather than numbers. Hence, what we are really doing is concatenating symbols in order to obtain the elements of the P_k's. Note that the symbol + is included in this concatenation each time strings of symbols are added together. For example, the element of P_3 above which appears in the second row and fourth column $(v_1v_2+v_5v_2+v_5v_3)$ is the concatenation of eight symbols. We interpret this string of symbols as follows: there are three paths of length three from vertex v_2 to vertex v_4, namely via vertices v_1 and v_2 or via vertices v_5 and v_2 or via v_5 and v_3. We observe that the basic operation of

Danielson's algorithm is the writing down of a single symbol. That is, in writing the element mentioned above, eight basic operations were involved. This process takes up more time and memory space than that required to perform a comparison of two integers which is the basic operation of our algorithm.

As was mentioned earlier, we are required to perform n nxn matrix multiplications in order to analyze an arbitrary graph for cyclic structure using Danielson's algorithm. Hence, we must generate n^3 elements, each of which may be a long string of symbols. Though some of these elements may be zero, at worst, as in the case where each pair of vertices is joined by an edge and a loop occurs at each vertex, all elements would be non-zero. Now, in the event we were to analyze such a graph, each element of P_k would contain n^2 terms of $(k-1)n^{k-1}$ symbols for the nodes and $n^{k-1} - 1$ plus (+) signs. Altogether, each of the n^2 terms of P_k would require

$$(k-1)n^{k-1} + n^{k-1} - 1 = kn^{k-1} - 1$$

symbols. Hence, if the basic operation is that of writing down a single symbol, then the number of operations required to locate all paths (and thus, cycles) in the worst case is given by

$$D = \sum_{k=2}^{n} \left(kn^{k-1} - 1 \right) .$$

To obtain an upper bound for the number of basic operations (i.e., comparison of integers) using our algorithm, we consider a special type of graph, namely the complete symmetric graph on n vertices. A digraph $G = G(V,E,g)$ is said to be <u>complete</u> if for every pair of vertices v_i and v_j in V, there exists an edge e in E such that either $g(e) = (v_i, v_j)$, or

$g(e) = (v_j, v_i)$. A digraph $G = G(V,E,g)$ is said to be symmetric if for every edge e such that $g(e) = (v_i, v_j)$ for v_i and v_j in V, there exists an edge e' such that $g(e') = (v_j, v_i)$. Thus, a complete symmetric graph on 4 vertices is shown in Figure 16 below:

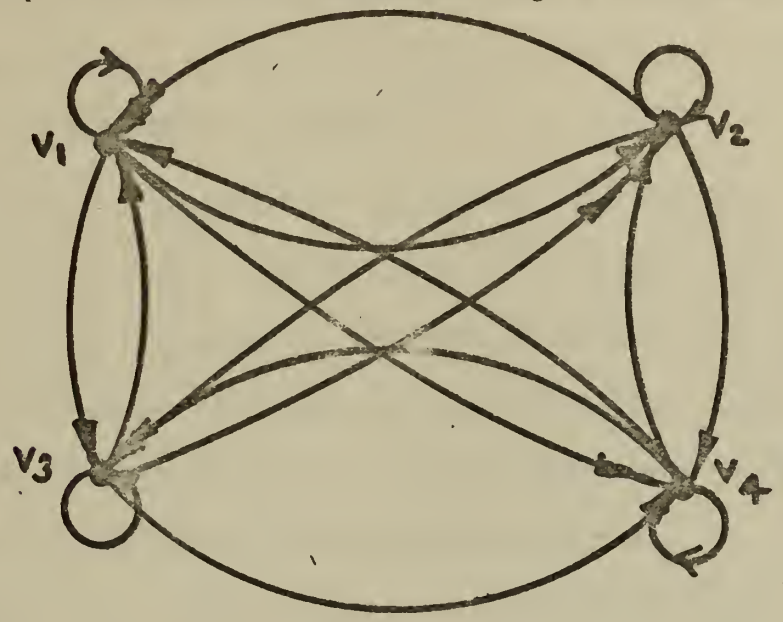


Figure 16. A Complete Symmetric Graph on 4 Vertices

We will compute the number of comparisons necessary to analyze this graph for cycles. Clearly this will be an upper bound on the number of comparisons required to analyze any digraph without parallel edges, since any other digraph with these properties will be a subgraph of the complete symmetric graph on the same number of vertices.

Note that the adjacency set of each vertex in a complete symmetric graph will contain n elements. Hence, at the first stage of the attainability tree (i.e., at the stage involving all vertices attainable by a path of length one from the root vertex), we have n elements which must be compared with one element, namely the root vertex. At this stage, we will locate one simple cycle: the loop which occurs at the root vertex. Thus, we continue on to the second stage with n(n-1) vertices. But, at this second stage, we will locate all of the other n-1 loops and all of

the simple cycles of length two which involve the root vertex. There are n-1 of these. Therefore, we continue on to the third stage with $(n(n-1) - (n-1) - (n-1))\, n = (n-1)(n-2)n$ vertices. Now, at this stage, we locate (n-1) (n-2) loops (obviously some are duplications), (n-1) (n-2) cycles of length two, and (n-1) (n-2) cycles of length three. Thus, we continue on to the fourth stage with

$$(n(n-1)(n-2) - (n-1)(n-2) - (n-1)(n-2) - (n-1)(n-2))n$$
$$= (n(n-1)(n-2) - 3(n-1)(n-2))n = (n-1)(n-2)(n-3)n$$

vertices. At the kth stage, we will have $(n-1)(n-2) \ldots (n-k+1)n$ vertices. Now, each of these must be compared with k vertices. Hence, at the kth stage, there are $kn(n-1)(n-2) \ldots (n-k+1)$ comparisons to make. We demonstrate this more formally in the following theorem.

THEOREM III. For a complete symmetric digraph (with loops at each vertex) on n vertices, there will be $(n-1)(n-2) \ldots (n-k+1)n$ vertices at the kth stage of the attainability tree.

Proof: The proof is by induction on k. Suppose k is 1. Then the number of vertices is n, which is clearly the case, since in the complete symmetric graph on n vertices, each element has n terms in its adjacency set.

Now, suppose the conclusion holds for k = p. That is, suppose that at the pth stage, there are $(n-1)(n-2) \ldots (n-p+1)n$ vertices. At this stage, we will locate $(n-1)(n-2) \ldots (n-p+1)$ loops, a like number of cycles of lengths 2, 3, ... , p. Hence, at the p+1 st stage, we will have

$$((n(n-1)(n-2) \ldots (n-p+1) - p(n-1)(n-2) \ldots (n-p+1))n$$
$$= n(n-1)(n-2) \ldots (n-p+1)(n-p)$$

vertices. Thus, since p was arbitrary, the conclusion holds in general.

We see, then, that an upper bound on the number of comparisons required to analyze an n-vertex digraph is given by

$$C = \sum_{k=1}^{n} kn(n-1)!/(n-k)!$$

Now, since n^{k-1} is clearly greater than $(n-1)!/(n-k)!$ for k greater than one, C will be less than D (for $n \geq 3$). In fact, as n becomes large, the difference between D and C becomes very great indeed. Also, since the complete symmetric graph occurs rarely in practice the number of comparisons will generally be very much less than the upper bound C (e.g., the graph in Figure 6 which required 79 comparisons as opposed to several million for the complete symmetric graph (with loops) on 14 vertices). However, since the non-zero elements of P_k will generally disappear quickly from P_k as k increases, the number of basic operations required to carry out Danielson's algorithm will often approach D. To illustrate, we will compare the number of basic operations required to analyze the graph in Figure 17(a) using Danielson's algorithm with the number of basic operations required to analyze the graph in 17(b) using our algorithm.

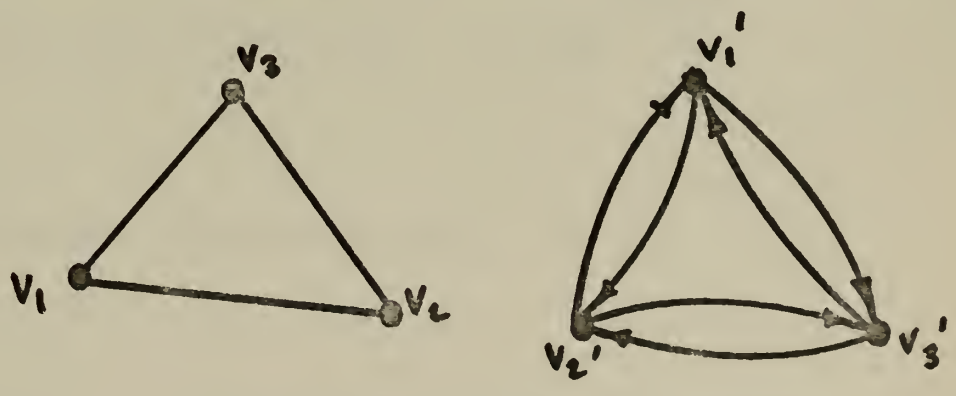


Figure 17 (a) An Undirected Graph (b) A Digraph

Using our algorithm on Figure 17(b), we obtain the attainability tree shown in Figure 18 (the only one for this graph, since all vertices appear in it).

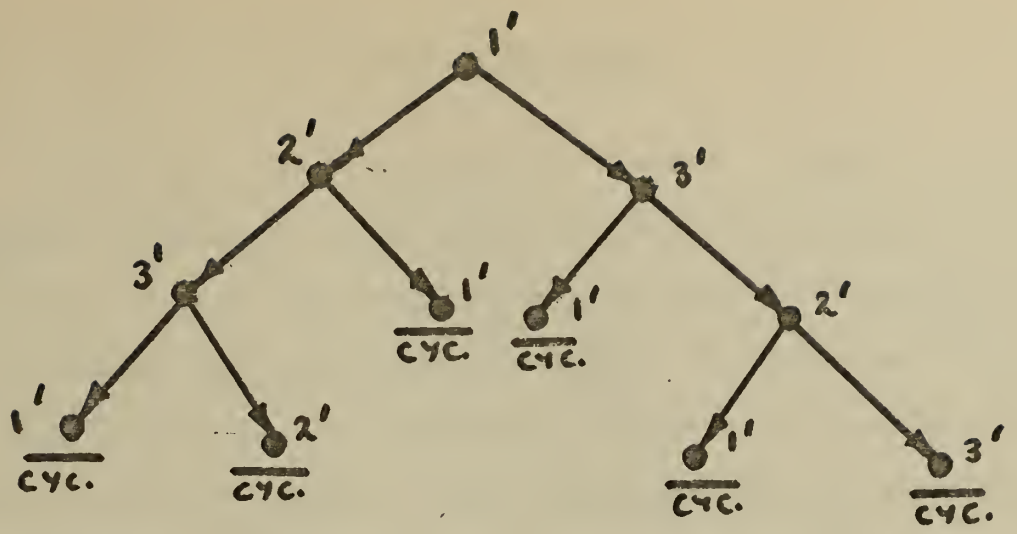


Figure 18. The Attainability Tree for the Graph of Figure 17(b)

We see that this graph requires 22 comparisons of integers.

Using Danielson's algorithm on the graph in Figure 17(a), we have

$$P_2 = \begin{bmatrix} v_2+v_3 & v_3 & v_2 \\ v_3 & v_1+v_3 & v_1 \\ v_2 & v_1 & v_1+v_2 \end{bmatrix}$$

and

$$P_3 = \begin{bmatrix} v_2v_3+v_3v_2 & v_2v_1+v_2v_3+v_3v_1 & v_2v_1+v_3v_1+v_3v_2 \\ v_1v_2+v_1v_3+v_3v_2 & v_1v_3+v_3v_1 & v_1v_2+v_3v_1+v_3v_2 \\ v_1v_2+v_1v_3+v_2v_3 & v_1v_3+v_2v_1+v_2v_3 & v_1v_2+v_2v_1 \end{bmatrix}.$$

We see that this required 78 basic operations.

VI. CONCLUSIONS

We have developed an algorithm for locating the simple cycles of a digraph which appears to be superior to previous algorithms in terms of the number of basic operations required and which is conservative in its use of memory space on the digital computer. In particular, we have analyzed our algorithm and the one originated by Danielson [3], and found that the upper bound for the number of basic operations using Danielson's algorithm was given by

$$D = \sum_{k=2}^{n} \left(k\, n^{k-1} - 1 \right) ,$$

while the upper bound on the number of basic operations using our algorithm was given by

$$C = \sum_{k=1} kn(n-1)!/(n-k)!$$

Since n^{k-1} is greater than $(n-1)!/(n-k)!$, we found that C was less than D. Moreover, since the basic operations required by Danielson's algorithm are more complex than those required by our algorithm, we see that our method represents a considerable improvement over the matrix technique.

By introducing the function p defined in Chapter IV, we were able to modify our algorithm for application to coalesced graphs. Thus, we can analyze extremely cumbersome graphs using the techniques developed, without being required to perform an undue number of basic operations.

It is our opinion that we have accomplished the goal we set out to achieve. However, it is believed that the algorithm might be made even more efficient. It was noted after the proof of Theorem II that some simple cycles may be located more than once. We suggest that there may be a way of eliminating this ambiguity. If this could be done, then the

number of basic operations required for certain graphs would be cut down considerably.

It was noted that Danielson's algorithm was designed for undirected graphs. Clearly, it is not very efficient. We suggest that a method similar to the one we have developed could be devised for application to undirected graphs which would be superior to Danielson's.

BIBLIOGRAPHY

1. Berge, C., Theory of Graphs and Applications, John Wiley & Sons, Inc., New York, 1962.

2. Busacker, R. G. and Saaty, T. L., Finite Graphs and Networks: An Introduction with Applications, McGraw-HillBook Company, New York, 1965.

3. Danielson, "On Finding the Simple Paths and Circuits of a Graph", I.E.E.E. Trans. Circuit Theory, V. CT 15: p. 294-295, Sept, 1968.

4. Harary, F., "Graph Theory and Electric Networks", I.R.E. Trans. Circuit Theory, V. 6: p. 95-109, May 1959.

5. Ore, O., Theory of Graphs, Colloquium Publications, V. 38, American Mathematical Society, Providence, R. I., 1962.

6. Reed, M. B. and Seshu, S., Linear Graphs and Electrical Networks, Addison-Wesley Publishing Company, Inc., Reading, Mass., 1961.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Assoc. Professor U. R. Kodres, Code 53Kr Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
4.	LTJG John M. Cochrane, USN 79 S. Sprague St., Coldwater, Michigan 49036	1

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
Naval Postgraduate School
Monterey, California 93940

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3 REPORT TITLE

On Locating the Simple Cycles in a Digraph

4 DESCRIPTIVE NOTES (Type of report and, inclusive dates)
Master's Thesis; June 1970

5 AUTHOR(S) (First name, middle initial, last name)

John Mackay Cochrane

6 REPORT DATE
June 1970

7a. TOTAL NO. OF PAGES
43

7b. NO. OF REFS
6

8a. CONTRACT OR GRANT NO.

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Naval Postgraduate School
Monterey, California 93940

13. ABSTRACT

An algorithm is stated for finding the simple cycles in a digraph which is believed to be superior to previous algorithms. The algorithm is stated in a way which lends itself to use on a digital computer. Suitable modifications are presented which allow the algorithm to be applied to coalesced graphs. Finally, the algorithm is compared to a previously used technique, and is shown to require fewer operations.

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Directed Graphs						
Cycle Locating Algorithm						